CDN-on-Demand: An Affordable DDoS Defense via Untrusted Clouds

Yossi Gilad
The Hebrew University of Jerusalem
mail@yossigilad.com

Amir Herzberg
Bar-Ilan University
amir.herzberg@gmail.com

Michael Sudkovitch
Bar-Ilan University
sudmike@gmail.com

Michael Goberman
Bar-Ilan University
micgob@gmail.com

***Abstract*—We present CDN-on-Demand, a software-based defense that administrators of small to medium websites install to resist powerful DDoS attacks, with a fraction of the cost of comparable commercial CDN services. Upon excessive load, CDN-on-Demand serves clients from a scalable set of proxies that it automatically deploys on multiple IaaS cloud providers. CDN-on-Demand can use less expensive and less trusted clouds to minimize costs. This is facilitated by the *clientless secure-objects*, which is a new mechanism we present. This mechanism avoids trusting the hosts with private keys or user-data, yet does not require installing new client programs. CDN-on-Demand also introduces the *origin-connectivity* mechanism, which ensures that essential communication with the content-origin is possible, even in case of severe DoS attacks.**

A critical feature of CDN-on-Demand is in facilitating easy deployment. We introduce the *origin-gateway* module, which deploys CDN-on-Demand automatically and transparently, i.e., without introducing changes to web-server configuration or website content. We implement CDN-on-Demand and evaluate each component separately as well as the complete system.

I. Introduction

The provision of efficient and Denial-of-Service (DoS) resilient web services has become an important goal for many websites. This goal is challenged by the unpredictability of demand for content, including the 'flash crowds' phenomenon, and by increasingly-powerful DoS attacks. One popular approach for dealing with this problem is to outsource content distribution to a *Content Delivery Network (CDN)*. CDNs deploy servers in various locations where they host proxies of websites, distributing computational and storage resources and delivering content to clients from a nearby server. CDNs use various mechanisms to provide scalable and robust services, including web-caches that reduce latency and communication volumes, and filters against clogging traffic. In particular, the dispersed servers and high-capacity connectivity of the CDN proved to mitigate *flooding* DoS attacks (e.g., see [31]).

However, for smaller sites the cost of a CDN service can be prohibitive. Indeed, despite their benefits, CDNs appear to be less popular among small and medium websites. In Figure 1 we present the usage distribution of commercial CDN providers among the 32K most popular websites (according to Alexa website-popularity rank [5]).[1] Line 1 of Figure 1 confirms earlier studies [24], [27]: most websites do not use CDNs, and furthermore, use of CDN declines for less-popular sites (from the 2000 place onwards). Note that larger organizations often use their own infrastructure instead of an external CDN provider.

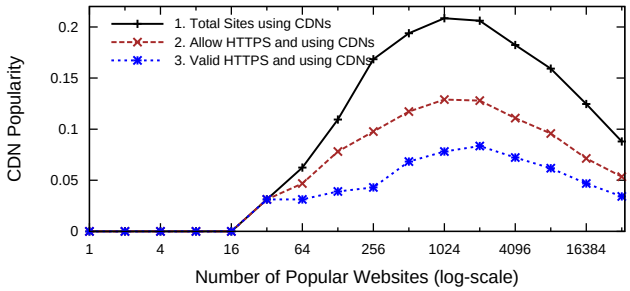


Fig. 1: CDN usage by website popularity. We count how many of X most popular sites use a CDN.

One reason for the limited use of CDNs is that smaller organizations may not be able to afford using CDNs on a regular basis, i.e., when not under attack, while temporal migration to a CDN introduces substantial administrative effort and financial costs. In particular, the CDN market is dominated by a few providers [24], [27], resulting in a less-competitive market and hence higher costs.

CDN-on-Demand offers an alternative approach for DoS-resiliency, which may be a better solution for small and medium websites. Namely, CDN-on-Demand is a *software* defense that automatically handles flash crowds and foils DoS attacks by managing flexible *cloud resources*, rather than a service, as offered by CDNs. Since CDN-on-Demand has an open design, its security, resiliency and performance properties can be carefully studied and improved, rather than forcing customers to rely on hard-to-validate claims of full-service providers which use proprietary mechanisms. Furthermore, websites can tailor the system to their needs, e.g., to ensure

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23109

[1] We use the method in [24] to identify whether a website uses a CDN: we query for its A and AAAA DNS records and check whether (1) the records point using canonical-name to a CDN, or (2) the website delegates DNS queries for its domain to a CDN's name server using an NS record; and (3) we retrieve the homepage of the website and check whether it obtains web-objects from one of the popular CDNs.

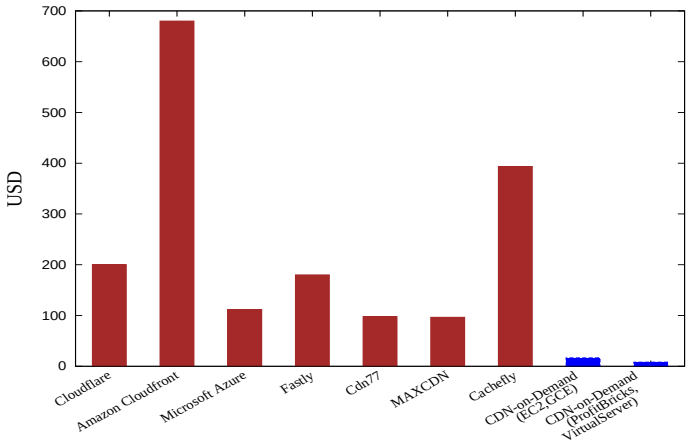


Fig. 2: Comparative evaluation of CDN service cost per month

desired user experience. In contrast, with service-based CDNs, even merely comparing and changing providers can be challenging.

To distribute content in face of strong DoS attacks, CDN-on-Demand deploys proxy servers on multiple Infrastructure-as-a-Service (IaaS) cloud providers, optimizing resource use to minimize expenses. Due to its flexibility, CDN-on-Demand can significantly reduce costs compared to 'full' CDN services; Figure 2 compares costs between CDN-on-Demand and several CDNs, including both widely-known CDNs and relatively inexpensive CDNs. The figure also compares the cost of *deploying CDN-on-Demand*, again comparing costs using well-known, reputed clouds (EC2 and GCE), to costs using less-known providers (ProfitBricks and VirtualServer). CDN-on-Demand costs are significantly lower, and use of the less-expensive providers can further save over half of the costs.

Less expensive cloud services, are typically less trusted, e.g., machines shared with other customers or less popular providers. In order to *securely* use such less expensive - and less trusted - providers, CDN-on-Demand does not require entrusting providers with the website's private key or certificate. This also solves an important concern with CDNs [24]. We achieve this property by protecting the website content, delivered to clients from CDN-on-Demand, using a novel *clientless secure-objects* mechanism.

Reducing trust in CDN proxies. Currently, the only content-security mechanism available using standard web clients is HTTPS (i.e., SSL/TLS), securing the communication channel between clients and proxies, but leaving the proxy with complete access to the content. Furthermore, Liang et al. [24] show that there are currently two options to support HTTPS connections using CDNs, both problematic. In the first option, the website shares its private key with the CDN; this presents challenges, for example when the website decides to switch to another CDN provider. In the second option, the website allows a certification authority to issue the CDN a certificate. As shown in [24], this approach is also vexed, since websites have difficulty revoking the certificate issued to the CDN. Note that even if the CDN operator is honest, one of its servers may be compromised; it may store copies of the data or keys in vulnerable locations, or deploy a web-server on the same host with an attacker's (allowing a variety of side-channel attacks, e.g., [34], [44]). We find that most websites using CDNs either do not support HTTPS connections or do not present valid certificates (compare line 1 to lines 2, 3 in Figure 1), illustrating the problem of securing communication when using CDNs.

Clientless secure-objects form a complementary mechanism to TLS/SSL, that provides security at the *HTTP object* level, i.e., each protected object is individually encapsulated (authenticated and, if needed, encrypted). Our design is *clientless*, namely it does not require changes to browsers and is *readily deployable* with today's websites. To support this mechanism without introducing changes to clients, our implementation uses a JavaScript agent that clients automatically download from the content-origin during their first connection to the website. Web object encapsulation, as an alternative or complementary mechanism to secure connections, was proposed several times since the very early days of web security (see [9], [13], [33], [36]). However, all proposals required installing browser extensions or helper applications to perform the security functions, e.g., validate authenticity of objects received, which proved a formidable deterrent to adoption (we provide an extensive discussion of related works at the end of the paper). Clientless secure-objects allow CDN-on-Demand to utilize cheaper untrusted providers and to switch between providers to optimize costs (without sharing the site's private key or certificate), while avoiding the deployment hurdles of introducing changes to clients (web-browsers).

Ensuring content-origin connectivity. CDNs offer several effective defenses against DoS. One basic defense is due to the fact that most of the website's content is static and cached by the CDN and therefore provided directly from the CDN's servers, which are difficult to attack (by being well connected, replicated and protected by filtering rules). CDN-on-Demand, similarly, leverages the robust connectivity and distributed infrastructure of IaaS cloud providers to efficiently deliver content. When necessary, e.g., under DoS, CDN-on-Demand 'scales-up' by using more proxy servers in the cloud to ensure good service.

However, some HTTP requests *require* communication with the content-origin, e.g., to support dynamic content or update web-objects stored on proxies. Such 'origin-bound requests' are vulnerable to additional DoS attacks, focused on the content-origin and its communication with the proxies. An attack might clog the connection between the CDN and the content-origin, blocking requests (or responses). The only defenses currently offered by CDNs against this threat are (1) to allow only traffic between the CDN and content-origin, dropping other packets, and (2) to use a secret, hidden IP address for the content-origin. These defenses are typically expensive and not always available. In particular, use of hidden IP address is often infeasible, especially to smaller websites of the kind we focus on, who wish to serve clients directly from their server when traffic load permits. Finally, these defenses often fail: attackers are often able to discover the 'hidden' IP address (see [40], [26]).

CDN-on-Demand includes the origin-connectivity mechanism, which ensures that the origin's server and website content remain reachable despite severe DDoS attacks. CDN-

on-Demand deploys manager servers that monitor performance of the content-origin; when loss-rates grow, the managers deploy proxies over IaaS clouds and modify the website's DNS mapping, directing clients to proxies. In order to ensure that communication between the content-origin and the proxies remains possible (to support dynamic content, updates and clientless secure-objects), CDN-on-Demand uses a different mechanism than suggested by today's CDNs: the system automatically establishes a clogging-resilient tunnel between the content-origin and CDN-on-Demand. This tunneling defense does not require hiding the content-origin's IP addresses. Instead, it performs efficient whitelist filtering using the proxy's IP addresses and randomized service port numbers, to allow only proxies to access the content-origin server, combined with redundant coding, to facilitate loss-tolerant communication with the content-origin.

Organization: Section II presents an overview of CDN-on-Demand. Section III describes the clientless secure-objects mechanism. Section IV describes the origin-connectivity mechanism. Section V describes the origin-gateway module, which facilitates deployment of CDN-on-Demand with existing websites. Section VI presents the proxy management mechanisms and Section VII evaluates CDN-on-Demand's performance and cost using our prototype. Sections VIII and Section IX present related works and our conclusions.

II. System Overview

This section presents an overview of CDN-on-Demand's design. We first describe the properties of the system, which match the goals outlined in the introduction. We then illustrate the deployment scenario, highlighting the system's components. Lastly, we describe the system's bootstrap mechanism, which facilitates its 'on-demand' property.

A. Properties

Robustness to DoS attacks. CDN-on-Demand ensures availability of all website content even when under severe DoS attacks, in particular, bandwidth-DoS attacks that attempt to clog communication to the website.

Secure use of untrusted providers. To allow maximal flexibility and reduce costs, CDN-on-Demand deploys over multiple low-cost IaaS-cloud providers. It ensures content-security against rogue providers or compromised cloud machines.

Efficient resource utilization. To establish an affordable system, CDN-on-Demand does not introduce significant overhead or costs when the website is not under attack or handling a flash-crowd. When the load on the content-origin server is too high, CDN-on-Demand automatically allocates the resources needed to provide sufficiently good service and frees them when unneeded.

Simple deployment and automated operation. CDN-on-Demand is easy to deploy and operates automatically, without requiring the site administrator to modify web-server configuration or website content. In particular, CDN-on-Demand does not require client-side installation and works with current IaaS-clouds infrastructure.

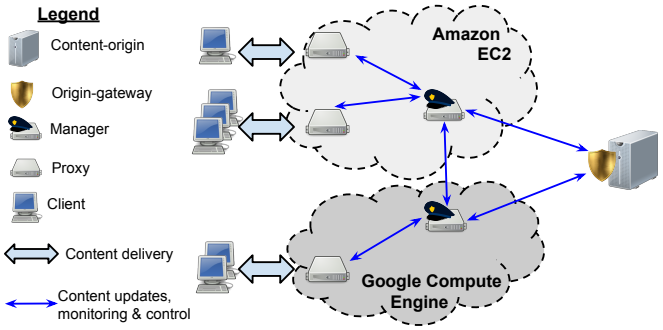


Fig. 3: CDN-on-Demand deployed over two clouds

B. Components and Deployment

Figure 3 illustrates CDN-on-Demand deployment, using two IaaS providers, Amazon Elastic Compute Cloud (EC2) and Google Compute Engine (GCE). The figure shows the main components involved in the system's operation: *managers and proxies*, which are deployed on IaaS-cloud instances, and the *origin-gateway*, which is deployed at the content-origin.

The managers perform four basic functions: *resource management, authoritative DNS, monitor/watchdog* and *second-level cache*. (1) The manager manages CDN-on-Demand's resources, by invoking and discarding CDN-on-Demand's cloud instances (for proxy services), to minimize costs while ensuring desired service level. To manage cloud instances, the manager receives (in configuration) the website administrator's credentials for using IaaS clouds. (2) The manager provides the authoritative DNS service for the content-origin's domain, mapping clients directly to the content-origin (under normal conditions, when CDN-on-Demand is dormant) or to CDN-on-Demand proxy nodes (when CDN-on-Demand is active, typically due to attack). (3) The manager monitors the availability of the content-origin, by running a 'watchdog' service (we describe this service as part of the bootstrapping process below). (4) Finally, the manager keeps an updated copy of all static, public objects in the content-origin, allowing CDN-on-Demand to provide this content even when the content-origin is under attack or otherwise unreachable. Notice that objects are stored using the clientless secure-objects mechanism, which ensures security even if cloud machines are corrupt.

For robustness, and to improve reliability and quality of monitoring, CDN-on-Demand supports deployment of more than one manager (as illustrated in Figure 3). In this case, one of the managers performs the resource management function; we refer to that manager as the *resource-manager*, and the others provide backup in case that manager fails. The other services, namely authoritative DNS, caching and origin-monitoring, are performed by all managers to distribute load and reduce latency. (DNS allows to specify multiple name-server addresses and proxies use the cache near them.)

The *origin-gateway* module is a software component that transparently implements the system's mechanisms at the content-origin's side, without requiring changes to the web-server or site's content. This simplifies deployment of CDN-on-Demand with existing websites. In contrast to proxies and managers, which are deployed on third party IaaS clouds,

typically on machines shared with other applications, the origin-gateway is deployed at the content-origin by the website administrator and handles client requests (or relays them to the content-origin). Hence, the origin-gateway is trusted with private keys and content.

C. System Bootstrap and Teardown

The managers' watchdog service periodically tests the content-origin's availability, by exchanging messages with the origin-gateway. These messages are exchanged over UDP, to avoid congestion control mechanisms (e.g., of TCP) and allow estimating the loss-rate and latency for communicating with the content-origin. Under normal conditions (i.e., no attack), watchdogs receive responses in a timely fashion. In this case clients connect directly to the content-origin, i.e., CDN-on-Demand is 'dormant' (proxies are not deployed), with negligible overhead and costs; the origin-gateway relays HTTP requests and responses between clients and the content-origin.

When the manager identifies that multiple responses are delayed or do not arrive (over a threshold), it activates CDN-on-Demand. The resource-manager deploys proxy servers, and changes the DNS mapping of the site's domain, directing clients to a nearby proxy (and also informs other managers, if deployed). While the system is active, the resource-manager adjusts the number of proxies based on traffic rates, in order to ensure sufficiently good service. When traffic rates are back to normal (e.g., below the capacity of the content-origin link), and the content-origin is available, the system goes back to the 'dormant' state. Namely, the manager changes the DNS mapping to point clients directly to the content-origin and removes the proxies. We describe the proxy-deployment and client-to-proxy mapping procedures in Section VI.

III. Clientless Secure-Objects

CDN-on-Demand is designed to work with any IaaS cloud provider to deploy proxies, in particular lower cost ones which are typically less trusted. Hence, we do not entrust the provider or cloud instances with the content-origin's private key or private user-data. In this section we introduce the *clientless secure-objects* mechanism, which ensures security for the data exchanged between the browser and CDN-on-Demand. This mechanism allows utilization of multiple IaaS providers, thus improving geo-coverage and reducing system cost, yet it does not necessitate trusting those providers and compromising on security. The clientless secure-objects mechanism is complementary to TLS/SSL, which we use to protect the communication links between clients, proxies and the content-origin.

We first describe the RootJS client-side agent, the key component of the clientless secure-objects mechanism, and present a simple yet important application of clientless secure-objects, which can be viewed as a simplified case of untrusted CDN: secure software downloads using mirror sites. We then describe the use of clientless secure-objects in CDN-on-Demand. We conclude this section by discussing the implementation of this module and its evaluation.

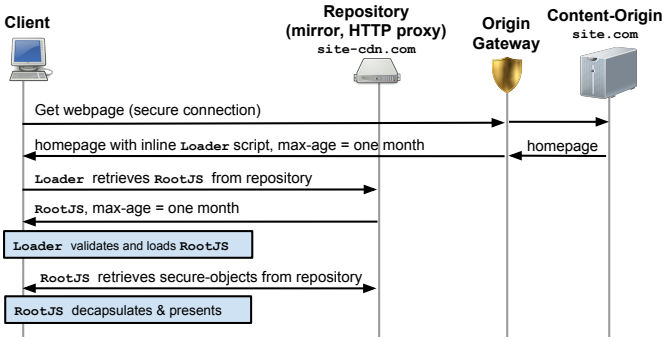


Fig. 4: Supplying the RootJS to new clients

A. Trusted Client-Side Agent

The *Root JavaScript* (RootJS) is the client-side agent of our system. This static and short (10KB) script is responsible for presenting web content to the user. In particular, the RootJS includes the website's public signature-verification key and validates the authenticity of objects that the client retrieves from untrusted repositories (mirror sites or CDN proxies).

Supplying the RootJS to new clients. Figure 4 illustrates a new client connecting to a website (site.com) that employs clientless secure-objects, securely loading the RootJS and then using it to display secure-objects. To ensure the authenticity of the RootJS, clients obtain a tiny Loader script from the content-origin over a secure (TLS) connection when they connect to the website for the first time. The origin-gateway 'injects' the Loader script into the website's HTML page in the content-origin response (i.e., 'inline script'); the script contains a hash function implementation (SHA1) and a hard-coded hash value of the RootJS. The Loader retrieves the RootJS from the repository, verifies the hash and then loads the script. The size of the Loader script is only 870Bytes; this is significant since the Loader is provided from the origin's site. To allow validation of objects stored in the repository, we send them with the Cross Origin Resource Sharing (CORS) header:

```
Access-Control-Allow-Origin: site.com
```

Specifying that the content-origin website (and only that website), site.com, may access objects from its repository, site-cdn.com.

When CDN-on-Demand is active, the manager (operating the authoritative DNS server) maps the website's domain name to proxy IP addresses. The client then opens a secure connection to the proxy machine in order to retrieve the website's homepage, which includes the Loader script and imports the RootJS. Since in our design the CDN does not hold the content-origin's private TLS key, it cannot handle the communication; the proxy merely relays the raw TLS communication to and from the content-origin. As we explain in Section IV, CDN-on-Demand's origin-connectivity mechanism leverages the system's control of both communication endpoints (proxies and origin-gateway) to establish a robust communication channel between them.

Caching, updating and revoking the RootJS. Because the RootJS is static, we use the browser's caching mechanism to

minimize latency and communication with the origin site in future connections. Specifically, the website's homepage with the Loader script and the RootJS are sent with the following HTTP header, to cache them at the client:

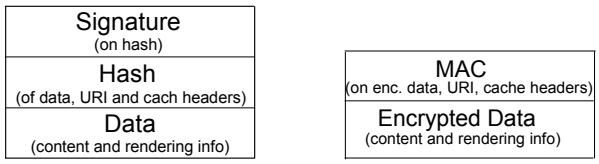
```
Cache-Control: public, max-age = one month
```

The clientless secure-objects mechanism allows forcing the Loader and RootJS out of the client's cache to facilitate code patches or revoke the website's private key if it was exposed. The mechanism works by distributing a small 'timestamp' object, via the repository, that approves use of the current RootJS version. This timestamp object is cached (both by the repository and at the client's browser) for a short period of time, much shorter than the RootJS (e.g., one day). The origin-gateway signs the 'timestamp' object and updates it on the repository (e.g., daily). To sign this sensitive object, the origin-gateway uses a dedicated private key (e.g., kept on an offline machine). The RootJS verifies the signature with the corresponding public key. If the client does not have the 'timestamp' object in the cache, the RootJS tries to fetch it from the repository. If it fails or the signature is invalid, then the RootJS circumvents the browser's caching mechanism (by specifying the Cache-Control: max-age = 0 HTTP header in the request) and retrieves a new version of the Loader from the content-origin as well as a new RootJS from the repository.

B. Securely-using Mirror Sites

Mirror sites may be viewed as a limited and inexpensive alternative to CDNs: a website peers with other sites to allow its clients to download files (typically software) from a server near them. Malicious or insecure mirror sites may modify the files hosted on their servers, e.g., to distribute malware. To address this threat, many websites using mirrors publish the hash of the download software file, and instruct users to validate authenticity of the file by computing the hash over its content and comparing with the published hash-value. In practice, however, only a small portion of savvy users perform such validation; many users do not, e.g., due to unwillingness or inability to install or use the validation tools [12]. We next explain how clientless secure-objects mechanism allows to protect users of software mirror sites. This simple application, limited to file downloads, helps understand the more involved application to CDN-on-Demand (distributing rich web content).

Instead of relying on users to validate the content of download files, the clientless secure-objects mechanism uses the RootJS to automate the validation. The content-origin signs the file together with an expiration date and its Uniform Resource Identifier (URI), which are 'linked' in the content-origin website. This ensures that a malicious mirror cannot change the file content, including change to obsolete version or another mirrored file. The website stores on the mirror the encapsulated version of the file, which includes the content (in plain-text), metadata such as URI, and a signature over all these fields. When users click on the download link at the website, they invoke the RootJS which fetches the file from the mirror site and validates its content. If successful, the RootJS calls the saveAs command which prompts to the user to save the original file's content.



(a) Public object (b) Private object

Fig. 5: Encapsulated objects (stored on repository)

C. Securely-using Untrusted Web Proxies

CDN-on-Demand ensures authenticity and confidentiality of web-content by storing encapsulated web-objects on its proxies. The origin-gateway performs the encapsulation automatically, when proxies retrieve web-objects from the content-origin (see details in Section V).

In order to support decapsulation of dynamic objects (AJAX), and avoid changing the original website, the RootJS 'hooks' the XMLHttpRequest's OnReadyStateChange method. Namely, the RootJS replaces the callback method that is invoked to handle dynamic objects to be the object-decapsulation method. If decapsulation is successful, the hook method passes the content to the original handler method, otherwise the RootJS discards the object.

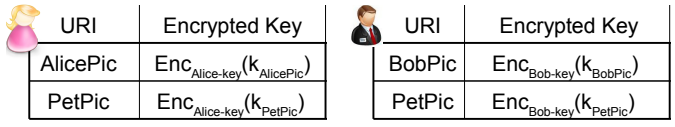
In the following two subsections, we discuss the encapsulation and rendering procedures, for public and then for private web-objects.

D. Public Objects

All users may access the website's public objects, which comprise much of the content in many sites such as online newspapers and stores. We ensure that public objects are not modified by malicious or compromised caching service. Figure 5a illustrates the encapsulated public object structure, which generalizes encapsulation of files stored on mirrors (see above) to support HTML rendering and HTTP caching options. Each encapsulated object specifies the parameters required to render its content (e.g., type and height/width parameters). The origin-gateway signs the object with its URI and HTTP caching directives ('Last-Modified', 'Expires' and 'max-age'). The content is stored in plain-text on the proxy along with the signature.

Decapsulation and Rendering. To display a web-page, the RootJS retrieves it from one of the CDN proxies, decapsulates it and validates its authenticity by verifying the content-origin's signature. Next, the RootJS displays the page to the user (by updating document.documentElement.innerHTML) and continues to retrieve the objects embedded in that page from the proxy, using the secure-objects' rendering information to display them on the page.

User Authentication. One particularly important public object is the *login form*, in which the user typically enters her credentials (e.g., username and password). Clientless secure-objects allow the client to securely retrieve the public login page from the CDN and send the user's credentials to the content-origin for verification. When the user authenticates, she receives from the origin-gateway her secret *user-key* and a signed cookie, which identifies the user (we provide details in



URI	Encrypted Key
AlicePic	$Enc_{Alice\text{-}key}(k_{AlicePic})$
PetPic	$Enc_{Alice\text{-}key}(k_{PetPic})$

(a) Alice's object table

URI	Encrypted Key
BobPic	$Enc_{Bob\text{-}key}(k_{BobPic})$
PetPic	$Enc_{Bob\text{-}key}(k_{PetPic})$

(b) Bob's object table

Fig. 6: Alice and Bob's object-tables. Alice and Bob share a picture of their pet (PetPic).

Section V). The user's key is stored at the client in local storage (available since HTML 5). This key is never shared with CDN proxies and allows the RootJS to display private content for that user (see next subsection). In contrast, the cookie is attached to all HTTP requests that the RootJS sends (by the browser), it allows the CDN proxies to enforce privacy policies in which only authorized users can retrieve private objects, as we next describe. The cookie also has an expiration date, which forces users to re-authenticate periodically.

E. Private Objects

Private objects are available only to a specific set of authorized users, therefore, we ensure confidentiality and authenticity of these objects. Private objects are encrypted and authenticated with symmetric *object-keys*. Figure 5b illustrates an encapsulated private object.

Each user is associated with an *object table*, illustrated in Figure 6, which maps the user's private objects' URIs to their keys. This allows users to share private objects by adding their keys to the corresponding users' object tables, i.e., without duplicating objects. The object-keys specified in the table are kept encrypted under the *user-key* which is not shared with the CDN (see discussion on user authentication above). Namely, the object table maps URIs to encrypted object-keys (see Figure 6). The content-origin caches copies of the users' object tables on CDN proxies. To present a private object, the RootJS retrieves the encrypted object and its encrypted object-keys from the proxy. Using the user's key, the RootJS decrypts the object's authentication and encryption keys, and then decapsulates and presents the private object.

F. Implementation and Evaluation

1) Implementation: Cryptographic computations introduce significant overhead in JavaScript (i.e., the RootJS). Measurements on a commodity mobile device (Samsung Galaxy S3) show that computations of SHA1 and AES-128bit require 2.1 and 3.4 milliseconds (ms) for processing a 100KB object, verifying an RSA-2048bit signature requires 14.5ms. Such overheads introduce noticeable delays to webpage load-time. However, the network round-trip time (RTT) is typically 10ms to 200ms, hence most of the time loading an object is spent waiting to receive its content. We optimize secure-object decapsulation by incrementally computing cryptographic operations: when the RootJS receives a data block, it processes that block while waiting for the remainder of the object's content to arrive.

Doing such incremental processing is simpler for private objects, which are protected by incrementally-computed, shared-key encryption and message-authentication mechanisms (e.g., AES and SHA1 process their input 'block-by-block'). To incrementally verify the signature over public objects, we take advantage of the common 'hash then sign' paradigm. To encapsulate, the origin-gateway computes the hash of the object to be signed, and sends it *at the beginning* of the encapsulated object, along with the signature (see Figure 5a). The RootJS receives the hash and signature prior to the object's content; this allows to validate that the given hash is properly-signed while receiving the rest of the content. The RootJS computes the hash of the content incrementally, as it arrives. When the response completes, the RootJS compares the result with the hashed value at the beginning of the object.

2) Empirical Evaluation with Real Web-Content: In the following set of experiments, we measured the overhead of the clientless secure-objects mechanism for handling and displaying content from popular websites. To obtain the web-content for this test we downloaded the homepages of the 2048 most popular websites according to Alexa [5] and the objects embedded in them. We stored these pages on a proxy server (deployed on EC2) and fetched them over HTTPS. We compare between their load times in three cases:

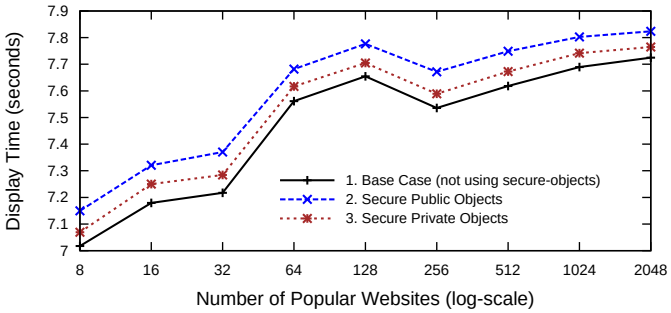
1) **Base Case:** we measure the time to load the homepages insecurely, i.e., without deploying clientless secure-objects.
2) **Public Objects:** we measure loading time when the homepages and objects within are encoded as secure public-objects, using SHA-1 and RSA2048.
3) **Private Objects:** we measure loading time when the homepages and objects within are encoded as secure private-objects, and protected using AES-128 and SHA1-HMAC. We assume that the user is logged-in (and hence has the user-key).

We measured these cases in two network scenarios: when the client connects to the Internet via Ethernet network, and when the client connects via cellular network (using LTE). The client machine is a PC with Core-i3 processor and 4GB of RAM, running Chrome browser (v44). Figure 7 illustrates our results.

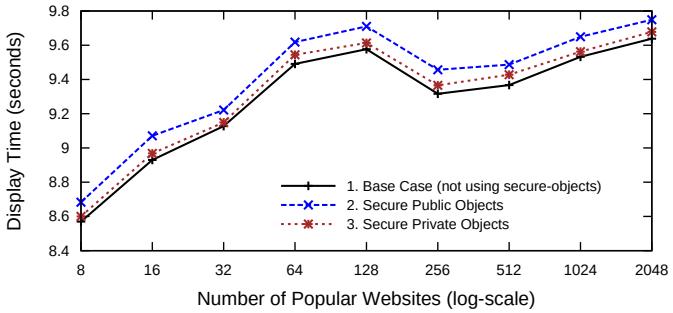
We observe that presenting a private web-page introduces 0.04%-0.2% delay to the display time, and presenting a public web-page introduces a higher overhead of 0.5%-1.8% delay to the display time. Notably, we find that the overhead in using clientless secure-objects is lower when the client connects via a cellular network, due to the increase in latency, which allows the RootJS to finish processing one block of the object's data before the next one arrives. These values are reasonable for most websites, and may be improved by using native cryptography and with further optimizations such as authenticating multiple public objects using Merkle trees (requiring one signature).

IV. Origin-Connectivity

CDN-on-Demand, like 'regular' CDNs, delivers static content from the proxies – which in turn, populate their cache by requesting new objects from managers (serving as a second-level cache; see description in Section II-B). Thus, for static objects, no communication is required with the content-origin itself. This significantly reduces the impact of DoS attacks



(a) Ethernet connection



(b) Cellular connection

Fig. 7: Clientless secure-objects, performance evaluation. The average display time for each of the X most popular sites.

on the content-origin. However, some Origin-to-CDN communication is essential, e.g., for delivering content-updates and providing dynamic services. In CDN-on-Demand, this communication channel has another critical role: it is required to securely distribute the Loader script which validates the RootJS (see Section III).

In the following subsections we describe and evaluate CDN-on-Demand's mechanisms ensuring resilient communication between the managers and the content-origin, in spite of clogging attacks on the content-origin: *dynamic whitelist filtering, loss-resilient tunnel* and *origin quotas*.

A. Dynamic Whitelist Filtering

In *whitelist filtering*, all packets sent over a congested link or router are dropped, with the exception of whitelisted packets. This is one of the most effective defenses against bandwidth-DoS (BW-DoS) attacks.

Traditional whitelist filtering is *static*. Namely, the customer provides to its ISP a list of legitimate source/destination IP addresses. The ISP then discards (filters) packets sent to that customer from other source addresses. Static whitelist filtering is one of the main anti-clogging defenses offered by current CDNs [10], [37]. It efficiently blocks *non-spoofed* bandwidth DoS attack packets, sent directly to the victim, as illustrated by Attacker 1 in Figure 8. In practice, non-spoofed attack packets are widely used in BW-DoS attacks, as they are easier to generate by attackers, since they can be sent by benign reflectors [29], [35], and clients running unprivileged malware and/or behind NATs or ingress-filtering routers.

However, static whitelisting may not prevent clogging by attackers able to send spoofed packets, illustrated by Attacker 2 in Figure 8. IP-spoofing attackers can often send significant amounts of traffic using fake source addresses, mainly by botnets; a recent study shows that about 15% of surveyed hosts are able to send spoofed packets [1], [7]. Although the amount of spoofed traffic that the attacker can send is therefore typically smaller than that of non-spoofed traffic, it may yet suffice to clog the limited link of the content-origin (a web-server of a small/medium site). The main defense currently adopted by CDNs against this threat, is to use a 'hidden IP address' for the content-origin, i.e., the address is known only to the CDN (e.g., SOS [22], CloudFlare [10], Akamai [2]). The hope is that attackers will not be able to clog the link

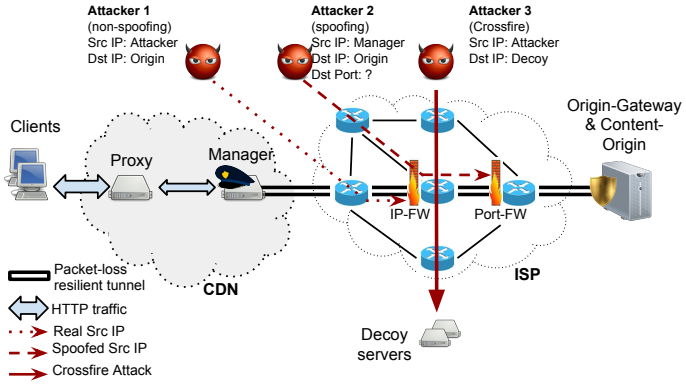


Fig. 8: Bandwidth-DoS attacks against the content-origin

to the content-origin, since they do not know its address. Recent works, however, show that these 'hidden' addresses can often be exposed by simple and efficient attacks [40], [26]. Furthermore, hiding the content-origin server address is not applicable to small sites, which want to use the *same IP address* when under attack (to communicate via the CDN), as when not under attack (to communicate directly with clients).

Hence, CDN-on-Demand does not depend on keeping the content-origin address hidden. Instead, to prevent clogging by spoofed packets, CDN-on-Demand uses *dynamic whitelist filtering*, using the source port and IP of the manager nodes, and the *destination port* of the content-origin (more specifically, of the origin-gateway); see illustration in Figure 9. The origin-gateway has a key shared with all managers, and each manager has a key shared with the origin-gateway (provided in configuration). Every refresh-interval τ (e.g., one hour), the origin-gateway selects a new service port by using its key to compute a pseudo-random function over the current time (in resolution τ, e.g., in hours). The managers perform a similar computation every τ interval to identify the new service port. Similarly, each manager also selects a new client port every τ interval, and the origin-gateway learns the new ports by computing the pseudo-random function using the managers' keys. After each change in ports there is a small grace period, where whitelist filtering allow both old and new ports. Using randomized ports to deploy filters against spoofing DoS attackers was studied in [6], [17]; we present experiments

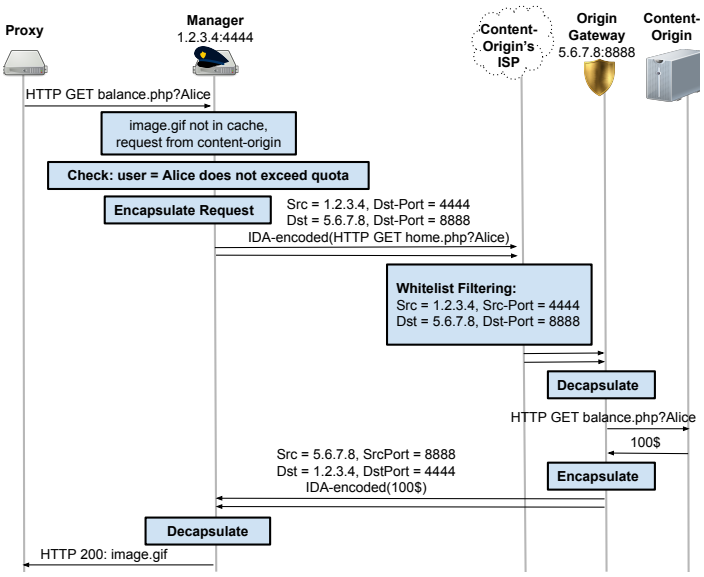


Fig. 9: Origin-connectivity mechanisms: dynamic whitelist-filtering, loss-resilient tunnel and quotas for dynamic queries.

showing effective deployment with CDN-on-Demand.

B. Loss-Resilient Tunnel

Many ISPs provide a whitelisting service, however this is not universal and the filtering service may introduce additional costs. Moreover, even when available, whitelist filters do not prevent attackers from clogging one of the links en-route to the content-origin with packets sent to other destinations, see Attacker 3 in Figure 8. This attack vector was introduced in [21], [38], and is considered difficult to filter (in particular using whitelists) since the victim is not the destination of the attack traffic. In this subsection we present our *loss-resilient tunnel* construction, a mechanism against DoS attacks which complements whitelist filters. The tunneling mechanism is deployed on the origin-gateway and CDN proxies; it is *independent* of third-party support such as the content-origin's ISP, which simplifies its deployment and allows to integrate with all sites.

Tunnel operation. The loss-resilient tunnel is invoked when the manager's origin-monitoring (i.e., watchdog) process detects significant loss-rates, which extensively reduce TCP's throughput; see Section II. Note that CDN-on-Demand proxies are deployed already at lower loss rates, i.e., usually earlier. Since the amount of traffic sent between the cloud nodes and the content-origin is very limited (as static content is served directly from the managers, deployed on the clouds), the tunnel is only required when the failure rate is high, as to cause TCP to seriously degrade or possibly fail.

In such case, the manager begins to use error-correcting codes to encode the traffic sent to the content-origin, typically HTTP requests. Complementary, if the origin-gateway identifies significant loss rates, it encodes the content-origin's responses to the managers; see illustration in Figure 9. Using error-correcting codes allows to recover communication despite high losses, as we next explain. The origin-gateway facilitates transparent deployment of this resilient-tunneling mechanism; we describe that in Section V.

The tunnel works at the transport layer, using UDP to send encoded data without waiting for acknowledgments. We use error-correcting encoding of m TCP packets into $n > m$ UDP packets. Specifically, we employ Rabin's Information Dispersal Algorithm [32], which proves to be efficient and allows recovery of all original m segments, as long as at least m encoded segments arrive.

The choice of m is relevant to performance, not to security, and depends on the network delay and transmission speed; in typical congestion situation, delays are high, and accordingly in our experiments we used $m = 100$. The number n of packets encoding the m input packets, is selected as a function of m and of the measured packet loss probability p. More precisely, senders calculate n such that the probability for losing more than $n - m$ packets is less than a pre-configured probability (e.g., 5%). Namely, for given channel loss rate p, we find the minimum n such that the probability that at least $n-m$ packets arrive is sufficiently large, e.g., over 95%; see Equation 1. (We approximate the sum in Equation 1 to the normal distribution using the central limit theorem to compute n efficiently).

$$\sum_{i=0}^{n-m} \binom{n}{i} p^i (1-p)^{n-i} \geq 0.95 \tag{1}$$

As we confirm in experiments below, the loss-resilient tunneling mechanism improves throughput significantly in case of attack causing high loss (despite the communication overhead). Since the tunnel is only established between the cloud and the content-origin, clients connecting to website need not change – they communicate with the proxies using standard web-protocols (HTTP/S over TCP).

One concern with the use of such mechanism, is that in response to packet loss, it increases transmission rates (by increasing redundancy), instead of slowing down (as done by TCP's congestion control mechanism). Hence, if the losses are result of 'regular' network congestion rather the DoS attack, this could increase (benign) congestion. To prevent this, the resource-manager maintains strict limits over the amount of traffic and requests sent, by each specific customer and in total; see next subsection. Furthermore, the traffic sent to and from the content-origin when CDN-on-Demand 'kicks-in' is much reduced (static content is cached, and provided to proxies from managers on the cloud), allowing (limited) redundancy in transmissions, without causing congestion.

C. Origin Quotas

The provision of static content from CDN proxies to clients significantly reduces the consumption of content-origin resources by *legitimate* clients. However, malicious clients may intentionally make bandwidth-consuming dynamic requests which cannot be provided by the proxies, and consume precious content-origin resources, in particular, bandwidth. Furthermore, attackers may send their requests for dynamic objects through multiple proxies, making it harder to identify an attack and utilizing higher bandwidth of multiple proxy nodes; see [39], [26].

To deal with such situations, and avoid excessive consumption of content-origin resources, CDN-on-Demand deploys *quotas*. Namely, the amount of resources consumed by each client is restricted. The details differ between *authenticated clients* and *new or sporadic clients*.

Authenticated users can be partially trusted, to be 'well-behaved' and not to launch DoS attacks against the content-origin. CDN-on-Demand identifies connections by such customers via the authentication cookie provided to the client from the content-origin upon authentication (see Section III-C). The resource-manager allocates a fair fraction of the maximal bandwidth it allows for communicating with the content-origin for each authenticated customer, up to a limit; see illustration in Figure 9.

To enforce quotas for new and sporadic users, CDN-on-Demand uses the client's IP address. As long as the total load on the content-origin is not too high, CDN-on-Demand allocates a limited amount of resources for each such client (limited per IP address). When the overall load on the content-origin resources is excessive, these clients cannot initiate any dynamic requests (priority given to authenticated users). CDN-on-Demand is still able to provide service to new and sporadic clients, even under high load conditions, or when their specific resource-requirements exceed a threshold, provided that their browser has a cached copy of the RootJS. Specifically, in this case, the manager will return a CAPTCHA which the RootJS will request the user to solve. Such mechanisms are well-studied, e.g., see [4], [25], [41]. [2]

D. Implementation and Evaluation

We implemented the defenses presented in CDN-on-Demand to ensure connectivity with the content-origin. We next present experimental evaluation, focusing on the dynamic filtering and loss-resilient tunnel mechanisms.

Setup. We deployed CDN-on-Demand using Amazon EC2 and Google Compute Engine IaaS clouds to deploy the system's proxies, and connected clients from different geographic regions using Planet-Lab machines [30]. We used a desktop machine to host the content-origin and connect it to the network via a 50Mbps link through another machine simulating its ISP which has 500Mbps link. We perform the experiments while our system handles 8K clients that repeatedly connect to the CDN and download a dynamic (non-cacheable) 50KB object from the website. To evaluate the origin-connectivity mechanisms, we use the manager to measure loss-rates when communicating with the content-origin, and use the clients to measure the response time from the CDN.

1) Dynamic Whitelist Filters: In this set of measurements we evaluate direct flooding attacks (i.e., Attackers 1 and 2 in Figure 8), where the attacker sends clogging traffic to the content-origin. We run the content-origin monitoring mechanism and employ the dynamic whitelisting mechanism (using dynamic IP-addresses and randomized service ports) presented above.

[2] An alternative to asking the users to solve a CAPTCHA, is for the RootJS to solve a Proof-of-Work (client puzzle), see e.g. [28]; however, this may benefit attackers who may use native code to quickly solve the puzzle, while legitimate clients use JavaScript.

Results. Lines 1 and 2 in Figure 10a show that by sending traffic to the content-origin, even at modest rates, the attacker can cause substantial packet loss rates and significantly increase the response time, eventually breaking most TCP connections between the manager and the content-origin. We observe a significant decline in throughput, e.g., throughput is only 2.45Mbps when the attacker transmits at 40Mbps (80% of the content-origin link capacity). We find that the whitelist filtering mechanism mitigates the direct attack vector; compare lines 1, 2 to lines 3, 4 in Figure 10a. In particular, the throughput of communication with the content-origin is 43.9Mbps, 18-times higher than without the filters. We note that the results of both spoofed and non-spoofed attack flavors are similar, hence we present their average in a single graph.

2) Loss Resilient Tunnel: In this set of experiments we assume that whitelist filtering is unavailable for the content-origin (e.g., not supported by its ISP) or that flooding traffic can circumvent whitelist filters deployed at the ISP (e.g., Attacker 3 in Figure 8), and evaluate the effect of a flooding DoS attack against the content-origin with and without the loss resilient tunnel defense. The length of encoded data blocks in the tunnel is $150KB$, encoding responses for 3 requests made via the manager (possibly by different proxies and clients); since network MTU is typically 1.5KB we get that $m = \frac{150\text{KB}}{1.5\text{KB}} = 100$.

Results. Our measurements, illustrated in Figure 10b, show that an attacker sending clogging traffic at over twice the content-origin's link capacity, can cause loss rates of over 84%, crippling TCP connections (see lines 1 and 2). The loss-resilient tunneling mechanism described above is enabled when loss rate is high (over 5%), to ensure sufficient quality of service. We observe with the tunneling mechanism installed, the loss-rate is kept steady below 5% and the response time moderately increases to 2.11 seconds (compare lines 1 and 2 to lines 3 and 4 in Figure 10b).

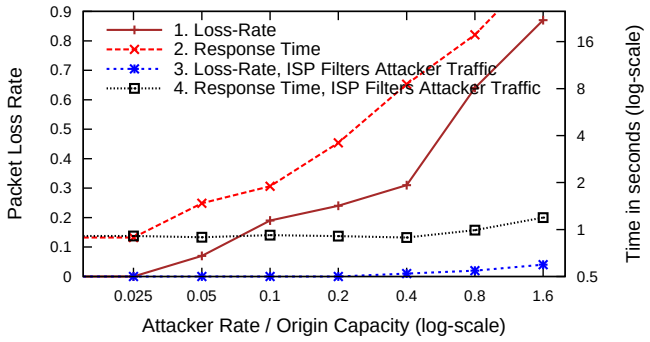
At the peak of the attack, causing high 84% loss-rate, the tunneling mechanism encodes every 100 packets to 640. Despite the overhead we find that this mechanism allows reasonable response time. This result is contrasted with normal TCP connections, which collapses when loss rates are significant.

V. The Origin-Gateway

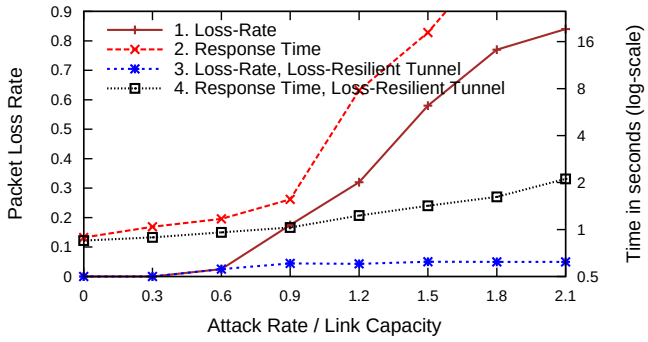
The origin-gateway is an easy-to-install module, acting as a gateway for the content-origin server; it facilitates transparent deployment of CDN-on-Demand's defense mechanisms without introducing changes to web-server configuration or website content. The origin-gateway has two main functions: First, the origin-gateway captures and converts web-objects that the content-origin sends into secure-objects (see Section III). Second, the origin-gateway implements the monitoring and resilient tunnel protocols at the origin's side (see Section IV). In this section we describe the operation and implementation of both origin-gateway functions.

A. Automated Conversion to Secure-Objects

The origin-gateway is a trusted component, deployed in the content-origin's network and managed by its administrator



(a) Dynamic whitelist filtering defense



(b) Loss-resilient tunnel defense

Fig. 10: Evaluation of DoS attacks against the content-origin

(cf. to CDN-on-Demand proxies). The administrator provides the origin-gateway with the website's private TLS key, and it serves as the TLS-endpoint at the origin side. Namely, it observes all requests and responses in clear-text. If CDN-on-Demand is dormant, i.e., there is no DoS attack, and the content-origin serves clients directly, then the origin-gateway only relays requests and responses, to and from the content-origin. In this case the origin-gateway only maintains a cache of encapsulated objects, in order to facilitate rapid transition to CDN-on-Demand when needed. We next describe the automated provisioning process of this cache.

To automatically encapsulate public objects, the origin-gateway periodically connects to the content-origin and 'crawls' the website (as an unauthenticated user). It identifies new public objects, retrieves and encapsulates them as described in Section III-C; see step 1 in Figure 11. When a request for one of the public objects arrives, the origin-gateway delivers the (encapsulated) secure-object in response. Namely, the origin-gateway acts as a cache; see step 2 in Figure 11.

Private objects, in contrast, are associated with specific users and are inaccessible to the origin-gateway while it crawls the website. To allow automated encapsulation of private objects, the administrator configures the origin-gateway with the name of the authentication cookie. The encapsulation mechanism for private objects is invoked when a GET request, which specifies an authentication cookie, arrives for an object not marked as public (identified in the 'crawling phase'). If the object is not already stored in the origin-gateway's cache, then the origin-gateway forwards the request to the content-origin; the origin-gateway also forwards the request when the object is cached but not associated with this user (i.e., object key is not in user's object-table), but then it changes the request type to HEAD, to only receive indication of success/failure. Namely, the origin-gateway uses the content-origin as an 'oracle' for checking whether the user is authorized to receive the object. If the response indicates success (HTTP 200 code), then the origin-gateway encapsulates the object for the user; see step 3 in Figure 11. We next describe the private-object encapsulation process performed by the origin-gateway.

The origin-gateway selects 'on the fly' a symmetric object-key (if the object does not already have one) and a user-key (if no key is already mapped to the user's authentication

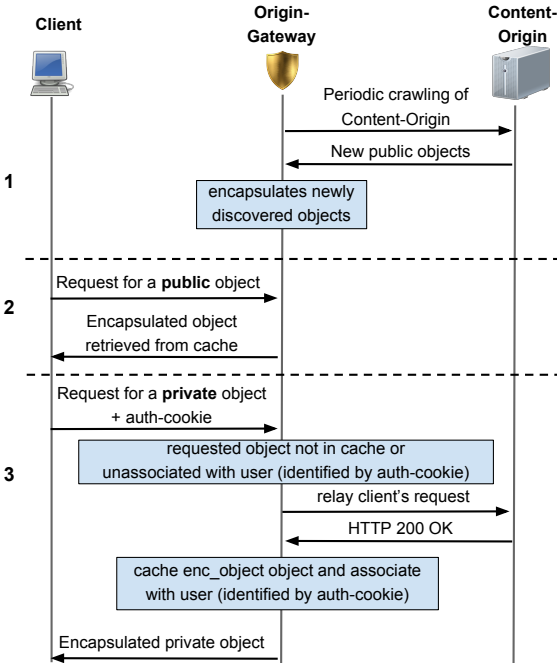


Fig. 11: Automated Conversion to Secure-Objects

cookie). The origin-gateway then encapsulates the object using the object's key and saves the secure-object in cache. Next, the object-key is encrypted with the user's key and saved in her object-table (see details in Section III-C). The origin-gateway also keeps track on the authentication cookie's expiration date; when the cookie expires and the user needs to re-authenticate, the origin-gateway deletes the user-key associated with the cookie and the corresponding object-table. When the user re-authenticates and receives a fresh cookie, the origin-gateway creates a new user-key and uses it to re-encrypt the user's private object-keys (as described above). This process, performed online, is very efficient: only symmetric cryptography is involved, objects are encrypted once and object-keys are only encrypted when the user authenticates.

Implementation. The origin-gateway keeps a cache of secure-

objects which maps an object's name to its encapsulated version and a flag that marks whether that object is public. In addition, for each user (identified by authentication cookie) the origin-gateway keeps an object-table, which maps the object name to the corresponding key (stored encrypted under the user's key). We use the BeautifulSoup library (v4.4) to crawl the content-origin website and identify public objects. To deliver content updates to CDN-on-Demand, the origin-gateway application uses the MitMProxy library (v0.13), which allows to register custom methods for processing HTTP requests and responses. The response-handler receives web-objects from the content-origin, encapsulates them and stores in its cache. If the object is private, then it also encrypts the object's key and saves it in the user's object-table.

B. Transparent Content-Origin Connectivity Module

When the origin-gateway identifies a significant loss rate p (e.g., $p > 5\%$), it begins encoding messages with redundancy, using the Information Dispersal Algorithm (IDA) [32] (see Section IV-B). Namely, the TCP communication between the origin-gateway and CDN-on-Demand's managers (deployed on the clouds) is encoded in UDP packets. The tunneled traffic carries redundant information to allow content-recovery in case of significant loss. The origin-gateway also implements the receiving-end of the loss-resilient tunnel, to decode IDA-encoded requests from the CDN-on-Demand managers. It decodes tunneled traffic (i.e., reconstructs the content despite loss) and then feeds the recovered (TCP) frames to the network stack, simulating arrival from the network.

Implementation. We use the NetfilterQueue library (v0.3) which allows to register methods for manipulating packets when they are emitted and received. We implement two methods: The first, *encoding-method*, captures TCP packets just before they leave the origin-gateway machine to one of the CDN-on-Demand managers (identified by their addresses). This procedure queues TCP segments until it aggregates m segments to encapsulate. The method then encodes these segments into n frames and sends them in UDP packets, all specifying an identifier that marks they were encoded together.

The second, *decoding-method*, captures tunneled traffic as it arrives (UDP traffic to the tunnel's service port). It queues the packets until it receives at least m out of the n packets with the same identifier to allow data recovery. The method decodes the data and recovers the underlying TCP communication, it then discards the UDP packets and inserts the recovered TCP segments into the TCP/IP handling stack instead.

VI. Proxy Selection and Placement

In Section II we described how and when the resource-manager decides to activate CDN-on-Demand and to serve clients from the proxies. In this section we address two related questions: how to map each client to a server, and where to deploy proxy servers.

A. Mapping Clients to Web-Servers

An efficient mapping between clients and proxies is key to reducing latency, which is an important benefit of CDNs. In the client-to-server mapping process, illustrated in Figure 12,

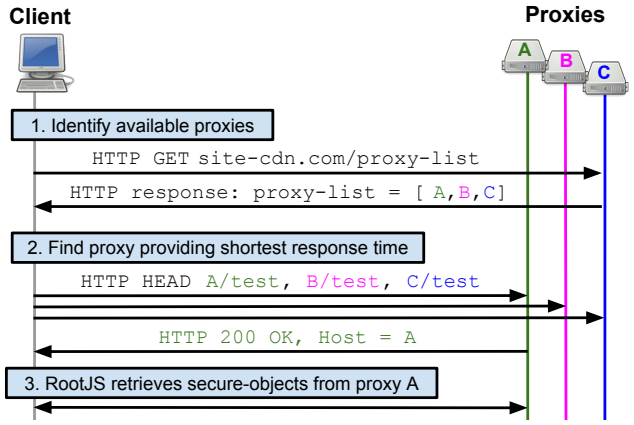


Fig. 12: Client-to-server mapping in CDN-on-Demand

each client finds and connects to the most responsive proxy in its region. To find the best proxy for a client we integrate a 'proxy-selection' module into the RootJS, which periodically retrieves the list of all active proxies in the client's geographic region (in parallel to handling the content from the web server).

The RootJS retrieves the list of available proxies from the CDN's domain, e.g., fetching https://site-cdn.com/proxy-list. If CDN-on-Demand is dormant, then that domain is mapped to the content-origin's IP address, the request reaches the origin-gateway who provides an empty list. In this case, the RootJS circumvents the clientless secure-objects mechanism, and loads content directly from the origin at site.com, over secure connection. If CDN-on-Demand is active, then the authoritative DNS servers (that managers run) map site-cdn.com to one of the proxies near the client. The client receives the proxy-list object from that proxy, which contains the addresses of available proxy web-servers (a non-empty list). The resource-manager updates the proxy-list when it deploys or decommissions a proxy (see the following subsection). Note that the private TLS key for the site-cdn.com is shared with the CDN (in contrast to site.com's key).

The RootJS evaluates the response time provided to the client by each proxy in the list (in case CDN-on-Demand is active) by sending a short HEAD request for a test object to each proxy and measuring the response time. Then, the RootJS begins using the first responding proxy. The RootJS caches the selected proxy for the next connections and periodically refreshes it by repeating the selection process.

B. Proxy Placement and Selection

We now explain how CDN-on-Demand manages the placement of proxy instances, i.e., where and which cloud instances to invoke, use and close down. The resource-manager monitors the utility of proxy machines in each region, using the cloud-provider APIs, which allow access to a variety of metrics such as CPU and network usage. We modify the number of proxies in a region when utility crosses a high/low threshold; in that case, the resource-manager executes the placement procedure to find the best cloud for hosting an additional proxy, or the best proxy to remove. For scale-up (adding a proxy), the procedure also evaluates the expected utility from

adding a proxy, and computes the utility/cost ratio; if beneath a threshold, the proxy is not added.

Placement procedure. We next describe the placement procedure upon a scale-up, i.e., deployment of additional proxy (scaling-down is performed in a similar fashion). To detect which cloud is the best for deploying the new proxy, the resource-manager temporarily powers-on one machine on each cloud for a brief time interval (one minute in our implementation). The candidate machines are not active proxies, but used to compare and select the best one. After the evaluation, each candidate machine either becomes an active proxy, or is turned off. Deployment of candidate machines for the brief evaluation step has a low cost; we measured less than 10^{-4}\$ per evaluation-step, with our CDN-on-Demand deployment.

During the evaluation step the active CDN proxies distribute the addresses of candidate machines to their clients, each client receives one address. Let C_m denote the set of clients participating in the evaluation of machine m. Through the RootJS, each client c compares the response time of the candidate machine m against the response time of the proxy that it uses. It then reports to the proxy $\delta(c,m) \geq 0$, which is the 'potential improvement'. This is essentially the response time of the proxy minus the response time of the candidate machine m (or zero, if m has longer response time). The resource-manager collects the clients' reports from the proxies; for each candidate machine m, it evaluates the average improvement in response time achieved by activating that machine, up to a maximal improvement δ_{MAX}, preventing one or few values from disproportionately influencing the outcome.

$$\Delta(m) = \frac{\sum_{c \in C_m} \min\{\delta_{MAX}, \delta(c,m)\}}{|C_m|} \qquad (\forall m, \Delta(m) \geq 0) \qquad (2)$$

The resource-manager then deploys a new proxy on the machine that provides the highest improvement in response time, provided that the improvement is over a minimum threshold, and powers-off all other candidate machines.

VII. Implementation and Evaluation

In this section we use a prototype implementation of CDN-on-Demand (deployed over two commercial clouds) to evaluate the performance and cost of the whole system. To motivate adoption of CDN-on-Demand and allow the community to validate our empirical measurements and results, we provide source-code of CDN-on-Demand in the project's website, https://autocdn.org. (Some modules are not yet available and would be added as soon as we complete a stable version.)

A. Setup

We deployed CDN-on-Demand over EC2 and GCE IaaS clouds with two managers (one employed as backup), see deployment in Figure 3. The managers were implemented in Python, configured to use EC2 and GCE APIs to manage the proxy machines running Squid HTTP proxy (v3.5). The content-origin server runs Nginx server (v1.9) and the origin-gateway is implemented by Python applications running on a Linux machine (Ubuntu Server 14.04.2). To simulate geographically-diverse clients accessing the website, we deployed 8K clients over Planet-Lab machines [30] located in

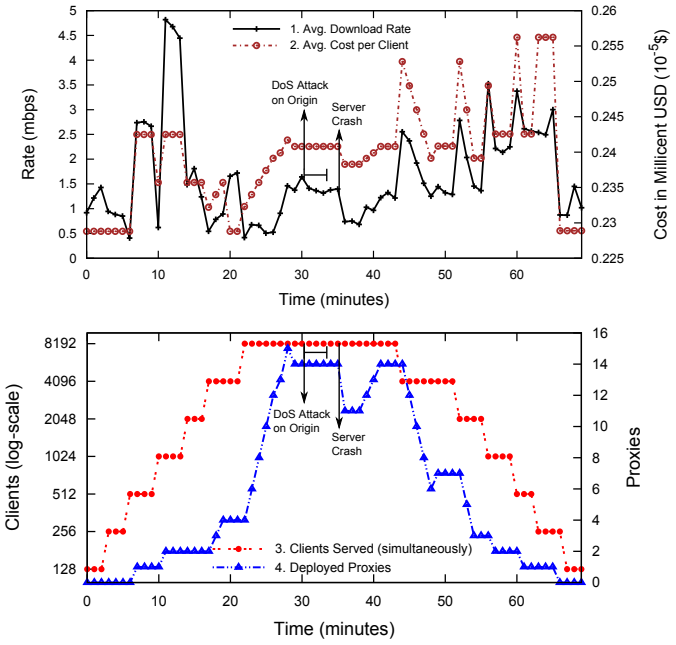


Fig. 13: CDN-on-Demand: Performance evaluation

different continents (each machine runs 16 clients). Each client repeatedly connects to CDN-on-Demand and downloads a web-page containing a 50KB image from the hosted website. The website's content (web-page and image) is cached by the proxies and managers for a configured one minute period; when these objects become stale, they request them again from the content-origin, simulating the content-update aspect of the CDN system.

B. Performance Evaluation

Figure 13 illustrates the scaling mechanism of CDN-on-Demand, dynamically adapting the system to changes in traffic volumes, client distribution, and server failures. We performed this set of experiments over a period of 70 minutes. We introduced events to the system every three minutes and measured the system's performance by comparing the traffic rates, cost, number of clients and number of proxies. We began the evaluation with the system serving the clients directly from the content-origin. At this point, the system cost is very low since the only chargeable instances running are the managers that monitor content-origin.

To test our system's resistance to flash crowds, we doubled the number of clients in the setup, six times, from 128 to 8192 (see lower part of Figure 13). We observe that CDN-on-Demand quickly detected the load and scaled-up accordingly (from no servers, i.e., only using content-origin, to 15 servers during peak load). This is also reflected by the momentary drops in the rate of service provided (top part of Figure 13).

Next, we initiated a BW-DoS (clogging) attack on the limited content-origin's link, for three minutes. We used 256 planet lab machines as zombies, sending UDP traffic to the content-origin IP address at full speed; total transmission rates were over 200mbps, well above the content-origin's link limit, 50mbps. Loss-resilient tunnels were automatically established to protect the communication between the content-origin and

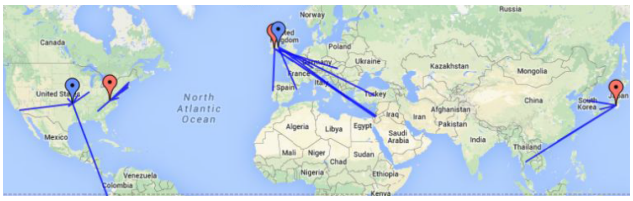


Fig. 14: CDN-on-Demand deployment and service snapshot. Each line represents connections with 16 clients. The system deploys proxies on GCE (blue pins) and EC2 (red pins).

the CDN (see details in Sections IV, V). We observed only a small effect on the average download rate (see Figure 13).

Following this, we simulated an IaaS provider failure, by abruptly powering-off all the proxies in one data-center, which served 12% of the clients. This caused an immediate decrease in the average transmission rate, followed by system recovery, automatically deploying new proxies in a near-by data-center.

We then reduced the number of connected clients (i.e., disconnected clients), from 8192 back to 128, again in six steps, to trigger the scale-down procedure and measure its impact. We observed that scaling-down the system decreases utility costs, but does not reduce the transmission rates. The reason is that most remaining clients still received service from nearby proxies.

Figure 14 illustrates the geographic distribution of clients and proxies. This system snapshot was taken 24 minutes into the experiment, when CDN-on-Demand deployed 5 proxies to serve 512 clients. The map illustrates the geographic coverage of CDN-on-Demand and shows that isolated clients connect to distant proxies, namely, CDN-on-Demand evaluates that deploying proxies near these clients will not significantly improve its performance.

C. Pricing: Survey and Comparison

In this subsection we evaluate the cost of CDN-on-Demand in typical scenarios and compare it with commercial CDNs.

Commercial providers offer a variety of pricing plans; in particular, some offer only fixed-price plans while others sell optional premium services such as DoS-protection. The service's cost varies according to the geographic location of deployed servers and connecting clients. These pricing models for committing service packages are contrasted against CDN-on-Demand, which leverages flexible IaaS clouds to deploy proxies only when needed, i.e., to handle flash-crowds or DoS attacks. In benign scenarios, CDN-on-Demand is dormant, employing only minimal number of machines to monitor the content-origin server, incurring negligible costs (see Section VI).

Methodology. We collected pricing data from several popular commercial CDNs and compared the costs of egress traffic (ingress traffic is typically free), fixed monthly fees and traffic filtering they provide. To estimate the cost of operating CDN-on-Demand we additionally account for the cost of operating cloud machines (managers and proxies). Under normal conditions (no attack/flash crowd), this cost is very low since the system is dormant; the cost increases only when the website is under heavy-load that causes CDN-on-Demand to scale-up.

CDN Provider	Out-Traffic (USD/GB)	Fixed fee (USD/month)	1TB/month Cost (USD/month)
Akamai	not published	not published	Refuses service to small sites
Cloudflare	0	200	200
Amazon Cloudfront	0.08	600	680
Microsoft Azure CDN	0.0725	39	111.5
Fastly	0.08	140	220
Cdn77	0.0395	58.25	97.75
MAXCDN	0.0575	39	96.5
Cachefly	0.2945	99	393.5

	Cloud Machine (USD/hour)	Out-Traffic (USD/GB)	Fixed fee (USD/month)	1TB/month Cost (USD/month)
CDN-on-Demand (on EC2 & GCE), no DoS/flash crowds	0.017	0.0619	0	**5.04**
CDN-on-Demand (on EC2 & GCE), DoS/flash crowds 5% of the time	0.017	0.0619	0	**16.605**
CDN-on-Demand (on ProfitBricks & VirtualServer), DoS/flash crowds 5% of the time	0.005	0.027	0	**8.136**

TABLE I: Cost comparison with CDN providers. Price quotes collected during June-July 2015.

It is challenging to compare costs between providers due to the varying pricing plans provided by each CDN; we therefore establish a comparable use-case. We evaluate the service cost per month, assuming a small/medium HTTPS website (i.e., requiring SSL/TLS support) that opts-in to using defenses against DoS attacks (if provided by the CDN), and serves 1TB of data to legitimate clients. For CDNs charging according to clients location, we leniently assume only the nearest, 'lowest-cost', clients connect (providing a lower bound for their price). We then evaluate cost of operating CDN-on-Demand in two scenarios: (1) a benign month without flash-crowds or DoS attacks (i.e., CDN-on-Demand is dormant), and (2) a month with DoS floods or flash crowds happening 5% of the time. We study these scenarios under two deployment cases: using EC2 and GCE – the two popular IaaS providers, and using ProfitBricks and VirtualServer which are cheaper less-popular ones.

Cost comparison. Our price survey, illustrated in Table I and in Figure 2, shows that CDN-on-Demand's price-tag in benign months is only a few USD and at least *one order of magnitude* lower than the 'next in line' commercial CDN services. In the second case, when the system handles flash crowds and DDoS attacks, CDN-on-Demand's cost is *5.8 times* lower than the following commercial CDN service when deployed on popular IaaS clouds (EC2 and GCE) or *12 times* lower when deployed over less popular ones (ProfitBricks and VirtualServer). We note that although some CDN providers advertise free DoS protection, there are complaints that these services are very limited and do not suffice to cope even with flash-crowds (e.g., see [11], [19]). Finally, we note that Akamai (one of the most popular providers) refuses to provide service to smaller websites which are the focus of this paper.

VIII. Related Works

We survey previous works in two categories, most related to the CDN-on-Demand architecture: (1) serving web clients from untrusted cache or proxy, and (2) using CDN and cloud services to cope with DoS attacks.

A. Secure Storage and Distribution

There were several proposals to allow secure websites to use CDNs without sharing their private keys.

SSL-Splitting [23] separates between encryption and authentication of SSL records. In SSL-Splitting, the CDN keeps objects in plain-text; the content-origin performs the SSL handshake with the client and provides only the negotiated encryption key to the CDN. For each SSL record, the CDN encrypts the data and the content-origin provides the authentication code, preventing the CDN from modifying the content. Since SSL-Splitting keeps the content-origin involved throughout the connection, it remains vulnerable to clogging DoS.

SINE [13] addresses this problem by having the content-origin compute an authentication tag for web-pages, rather than SSL-records. The client retrieves the tags from the content-origin over a secure (HTTPS) connection and the web-pages from the CDN. In SINE, the content-origin is only involved at the beginning of the connection for sending authentication tags. In HTTPi [36] authentication tags are attached to web-objects to allow object-caching while securing transmissions over HTTP. Choi et al. [9] suggest an alternative protocol to HTTPS; their protocol only provides authentication, and requires changes to the current Internet caching mechanisms.

All of these proposals [9], [13], [36], [23] introduce modifications to clients (web-browsers), for parsing and verifying the authentication tags before presenting web-content. Such changes usually involve significant deployment efforts and will force websites to support legacy clients even after the changes have been accepted to major browsers.

Recent work on Information-Centric Networking (ICN) ties security to objects. In particular, the Named Data Networking paradigm [43], a part of the ICN efforts, suggests that the sender signs all web-objects that it sends [15]. However, supporting ICN requires major changes to Internet infrastructure, see [3], [16]. In contrast to these works, the clientless secure-objects mechanism establishes a trusted JavaScript agent (the RootJS), thereby avoiding any changes to clients' browsers or to CDN/cloud infrastructure. As we show in experiments, clientless secure-objects provide a practical and lightweight implementation for some ICN paradigms.

Liang et al. studied the current practices of supporting HTTPS websites in CDNs [24]. They found that most CDNs use the website's private key to handle HTTPS clients. . To remedy these issues, they suggest using DANE [18], a recently proposed DNS extension, relying on DNSSEC and requiring changes to the clients' certificate validation mechanisms. These are formidable obstacles.

B. CDN-like DoS Defenses using Clouds

CDN-on-Demand uses low-cost, untrusted IaaS-clouds, to protect websites against DoS attacks. The idea of using IaaS clouds as a lower-cost alternative to fully-managed CDN services was proposed by Broberg et al. [8]. They proposed to use storage clouds as a lower-cost alternative to the basic CDN services, i.e., to provide services using multiple machines located closer to the clients. However, Broberg et al. did not address DoS attacks or other security challenges. In fact, their cost-savings reflect simply the high premium of CDN providers at the time. In contrast, CDN-on-Demand allows significant cost savings by using cloud resources only when necessary, to handle DoS attacks or flash-crowds.

The use of clouds to provide scalable DoS-prevention services was proposed in several works. Most of these works focus on using clouds to filter different types of DoS traffic; see [42] for a comprehensive treatment of this subject, also providing extensive bibliography. Some works, such as [20], take a different approach and migrate the service (permanently) to the cloud, reassigning tasks to different cloud nodes to deal with DoS attacks. This work takes a different 'on-demand' approach, and therefore focuses on different problems: ensuring content security via untrusted providers, and mitigating BW-DoS attacks on the communication with the content-origin.

There are many proposals and commercial products for mitigating bandwidth-DoS (clogging) attacks, a basic challenge in ensuring Internet availability, see survey in [14]. CDN-on-Demand mechanisms for ensuring connectivity to the content-origin build on some of these ideas. First, we facilitate DoS defenses by routing via the cloud-nodes, essentially creating an overlay network; this is related to many works on mitigating DoS attacks using overlay routing, e.g., [22]. Second, we use very efficient port-based filtering of spoofed traffic, extending the technique in [6].

IX. Conclusions and Future Work

Website operators face many challenges in ensuring efficient, low-latency, high-availability, and secure service to globally dispersed users. In particular, network conditions may change rapidly, most notably due to flash-crowds and DoS attacks. CDNs typically meet these challenges by offering automated and scalable content distribution.

However, many sites cannot afford CDN services and prefer to use one or several self-managed servers or cloud machines. This option offers reduced costs under normal conditions, but does not address DoS attacks.

To tackle this challenge we presented CDN-on-Demand, an automated system allowing websites, even small, to mitigate powerful DoS attacks with minimal operational costs and easy, automated deployment. CDN-on-Demand uses Infrastructure-as-a-Service (IaaS) cloud services to deploy proxy servers, but only when required. CDN-on-Demand does not require trusting the IaaS providers with long-term keys and secrets, and mitigates attacks targeting the essential communication between the content-origin and the cloud nodes.

Future Work. We leave several directions open for research. First, extending the clientless secure-objects to mitigate traffic analysis attacks where proxies learn the set of users sharing a private object or when they access it. Second, the performance of public object decapsulation may be improved, e.g., by encapsulating several objects on the same page under one

signature. Third, a kernel module implementation of the loss-resilient tunnel application at the origin-gateway may allow to improve performance under saturating conditions.

ACKNOWLEDGMENTS

This work was supported by grant 1354/11 from the Israeli Science Foundation (ISF), and by grants from the Check Point Institute for Information and Security (CPIIS) and the Ministry of Science, Technology and Space, Israel.

REFERENCES

[1] ADVANCED NETWORK ARCHITECTURE GROUP. Spoofer Project. http://spoofer.csail.mit.edu, May 2015.

[2] AFERGAN, M., ELLIS, A., SUNDARAM, R., AND RAHUL, H. Method and System for Protecting Web Sites from Public Internet Threats, Aug. 21 2007. US Patent 7,260,639.

[3] AHLGREN, B., DANNEWITZ, C., IMBRENDA, C., KUTSCHER, D., AND OHLMAN, B. A Survey of Information-Centric Networking. *IEEE Communications Magazine 50*, 7 (2012), 26–36.

[4] AHN, L. V., BLUM, M., HOPPER, N. J., AND LANGFORD, J. CAPTCHA: Using Hard AI Problems for Security. In *EUROCRYPT* (2003), Springer-Verlag, pp. 294–311.

[5] ALEXA WEB INFORMATION COMPANY. Top Sites. http://www.alexa.com/topsites, 2015.

[6] BADISHI, G., HERZBERG, A., AND KEIDAR, I. Keeping Denial-of-Service Attackers in the Dark. *IEEE Transactions on Dependable and Secure Computing 4*, 3 (2007), 191–204.

[7] BEVERLY, R., KOGA, R., AND CLAFFY, K. Initial Longitudinal Analysis of IP Source Spoofing Capability on the Internet. Internet Society Article, July 2013.

[8] BROBERG, J., BUYYA, R., AND TARI, Z. MetaCDN: Harnessing Storage Clouds for High Performance Content Delivery. *Journal of Network and Computer Applications 32*, 5 (2009), 1012–1022.

[9] CHOI, T., AND GOUDA, M. G. HTTPI: An HTTP with Integrity. In *Computer Communications and Networks (ICCCN)* (2011), IEEE, pp. 1–6.

[10] CLOUDFLARE INC. CloudFlare Advanced DDoS Protection. https://www.cloudflare.com/ddos.

[11] EASTDAKOTA. How Much Traffic is Too Much Traffic For CloudFlare? Online at Hacker News, Jan 2013.

[12] FU, K., KAASHOEK, M. F., AND MAZIERES, D. Fast and Secure Distributed Read-only File System. In *Symposium on Operating System Design & Implementation* (2000), USENIX Association, pp. 13–34.

[13] GASPARD, C., GOLDBERG, S., ITANI, W., BERTINO, E., AND NITA-ROTARU, C. SINE: Cache-Friendly Integrity for the Web. In *Workshop on Secure Network Protocols* (2009), IEEE, pp. 7 – 12.

[14] GEVA, M., HERZBERG, A., AND GEV, Y. Bandwidth distributed denial of service: Attacks and defenses. *IEEE Security & Privacy 12*, 1 (2014), 54–61.

[15] GHALI, C., TSUDIK, G., AND UZUN, E. Network-Layer Trust in Named-Data Networking. *Computer Communication Review 44*, 5 (2014), 12–19.

[16] GHODSI, A., SHENKER, S., KOPONEN, T., SINGLA, A., RAGHAVAN, B., AND WILCOX, J. Information-Centric Networking: Seeing the Forest for the Trees. In *HotNets* (2011), ACM, pp. 1–6.

[17] GILAD, Y., AND HERZBERG, A. LOT: A Defense Against IP Spoofing and Flooding Attacks. *ACM Transactions on Information and System Security 15*, 2 (July 2012), 6:1–6:30.

[18] HOFFMAN, P., AND SCHLYTER, J. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698 (Proposed Standard), Aug. 2012.

[19] HOSTGATORHOST. Cloudflare Free Plan. Web Hosting Talk, Mar 2014.

[20] JIA, Q., WANG, H., FLECK, D., LI, F., STAVROU, A., AND POWELL, W. Catch Me If You Can: A Cloud-Enabled DDoS Defense. In *International Conference on Dependable Systems and Networks* (June 2014), pp. 264–275.

[21] KANG, M. S., LEE, S. B., AND GLIGOR, V. D. The Crossfire Attack. In *IEEE Symposium on Security and Privacy* (2013), IEEE Computer Society, pp. 127–141.

[22] KEROMYTIS, A. D., MISRA, V., AND RUBENSTEIN, D. SOS: Secure Overlay Services. In *SIGCOMM* (2002), vol. 32, 4 of *Computer Communication Review*, ACM Press, pp. 61–72.

[23] LESNIEWSKI-LAAS, C., AND KAASHOEK, M. F. SSL Splitting: Securely Serving Data from Untrusted Caches. *Computer Networks 48*, 5 (2005), 763–779.

[24] LIANG, J., JIANG, J., DUAN, H., LI, K., WAN, T., AND WU, J. When HTTPS Meets CDN: A Case of Authentication in Delegated Service. In *IEEE Symposium on Security and Privacy* (2014).

[25] LIU, X., YANG, X., AND LU, Y. To Filter or to Authorize: Network-Layer DoS Defense Against Multimillion-Node Botnets. In *SIGCOMM* (2008), ACM, pp. 195–206.

[26] MIU, T. T., HUI, A. K., LEE, W., LUO, D. X., CHUNG, A. K., AND WONG, J. W. Universal DDoS Mitigation Bypass. *Black Hat USA* (2013).

[27] NEUSTAR. State of DDoS Protection, 2012.

[28] PARNO, B., WENDLANDT, D., SHI, E., PERRIG, A., MAGGS, B. M., AND HU, Y.-C. Portcullis: Protecting Connection Setup from Denial-of-Capability Attacks. In *SIGCOMM* (2007), J. Murai and K. Cho, Eds., ACM, pp. 289–300.

[29] PAXSON, V. An Analysis of Using Reflectors for Distributed Denial-of-Service Attacks. *Computer Communication Review 31*, 3 (2001), 38–47.

[30] Planet-lab. http://www.planet-lab.org/.

[31] PRINCE, M. The DDoS That Almost Broke the Internet. CloudFlare Blog, April 2013.

[32] RABIN, M. O. Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance. *Journal of the ACM 36*, 2 (1989), 335–348.

[33] RESCORLA, E., AND SCHIFFMAN, A. The Secure HyperText Transfer Protocol. RFC 2660 (Experimental), Aug. 1999.

[34] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Computer and Communications Security* (2009), ACM, pp. 199–212.

[35] ROSSOW, C. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *NDSS* (February 2014).

[36] SINGH, K., WANG, H. J., MOSHCHUK, A., JACKSON, C., AND LEE, W. Practical End-to-End Web Content Integrity. In *WWW* (2012), ACM, pp. 659–668.

[37] SITARAMAN, R. K., KASBEKAR, M., LICHTENSTEIN, W., AND JAIN, M. Overlay Networks: An Akamai Perspective. In *Advanced Content Delivery, Streaming, and Cloud Services*. John Wiley and Sons, 2014.

[38] STUDER, A., AND PERRIG, A. The Coremelt Attack. In *ESORICS* (2009), vol. 5789 of *LNCS*, Springer, pp. 37–52.

[39] TRIUKOSE, S., AL-QUDAH, Z., AND RABINOVICH, M. Content Delivery Networks: Protection or Threat? In *ESORICS* (2009), vol. 5789 of *LNCS*, Springer, pp. 371–389.

[40] VISSERS, T., GOETHEM, T. V., JOOSEN, W., AND NIKIFORAKIS, N. Maneuvering Around Clouds: Bypassing Cloud-based Security Providers. In *CCS* (2015), ACM.

[41] VON AHN, L., MAURER, B., MCMILLEN, C., ABRAHAM, D., AND BLUM, M. reCAPTCHA: Human-based Character Recognition via Web Security Measures. *Science 321*, 5895 (2008), 1465–1468.

[42] YU, S. *Distributed Denial of Service Attack and Defense*. Briefs in Computer Science. Springer, 2014.

[43] ZHANG, L., ESTRIN, D., BURKE, J., JACOBSON, V., THORNTON, J. D., SMETTERS, D. K., ZHANG, B., TSUDIK, G., MASSEY, D., PAPADOPOULOS, C., ET AL. Named Data Networking (NDN) Project, 2010.

[44] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM Side Channels and Their use to Extract Private Keys. In *CCS* (2012), ACM, pp. 305–316.